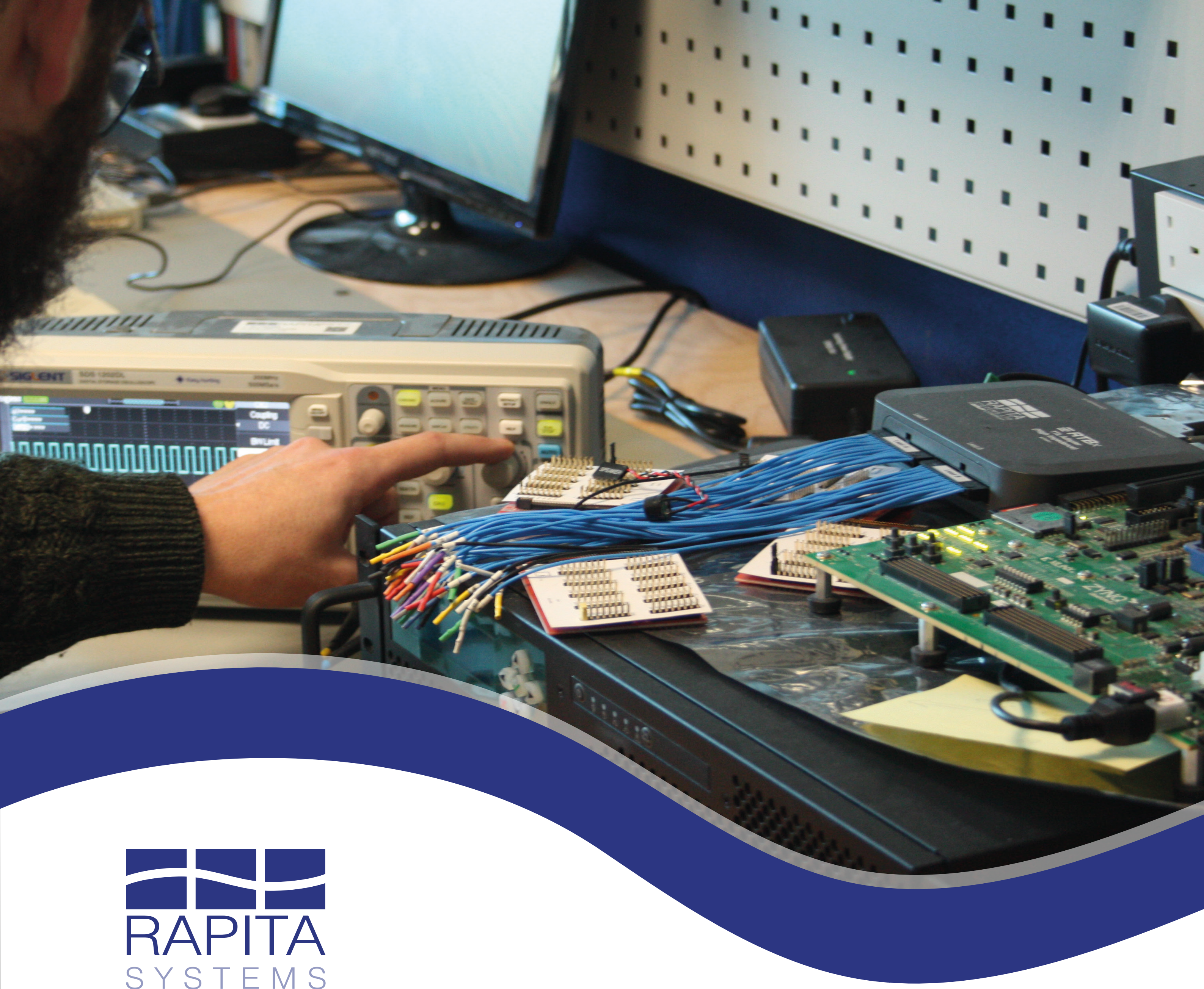


RAPITA SYSTEMS

A DANLAW COMPANY

Safety through quality

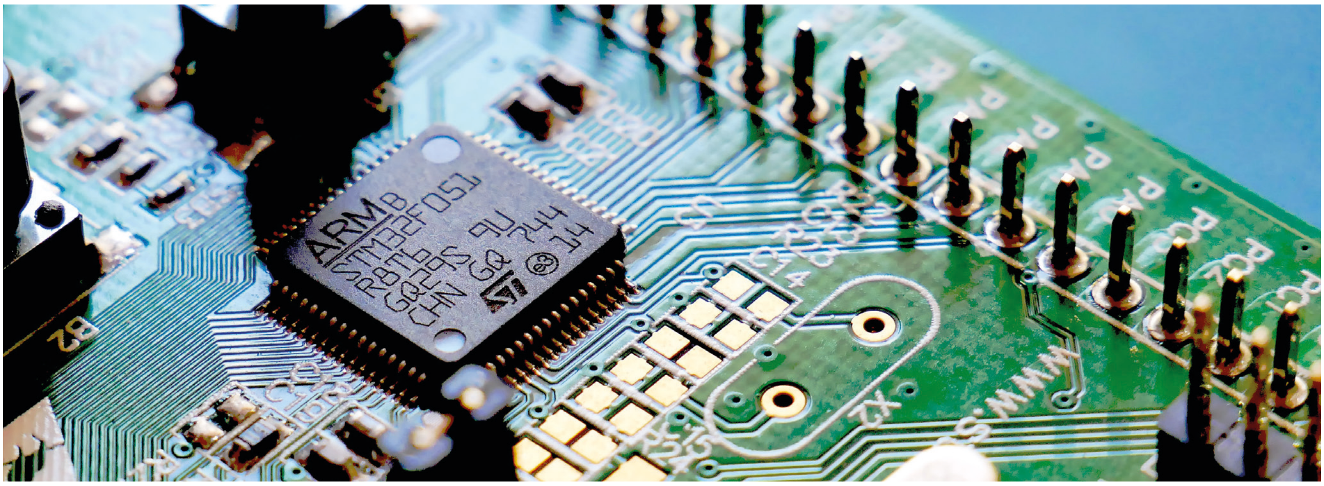
WHITE PAPER

5 key factors to consider when selecting an embedded testing tool

Document ID: MC-WP-012 5 factors embedded testing v4 INDT-v10

Contents

1. Why is embedded software testing important?

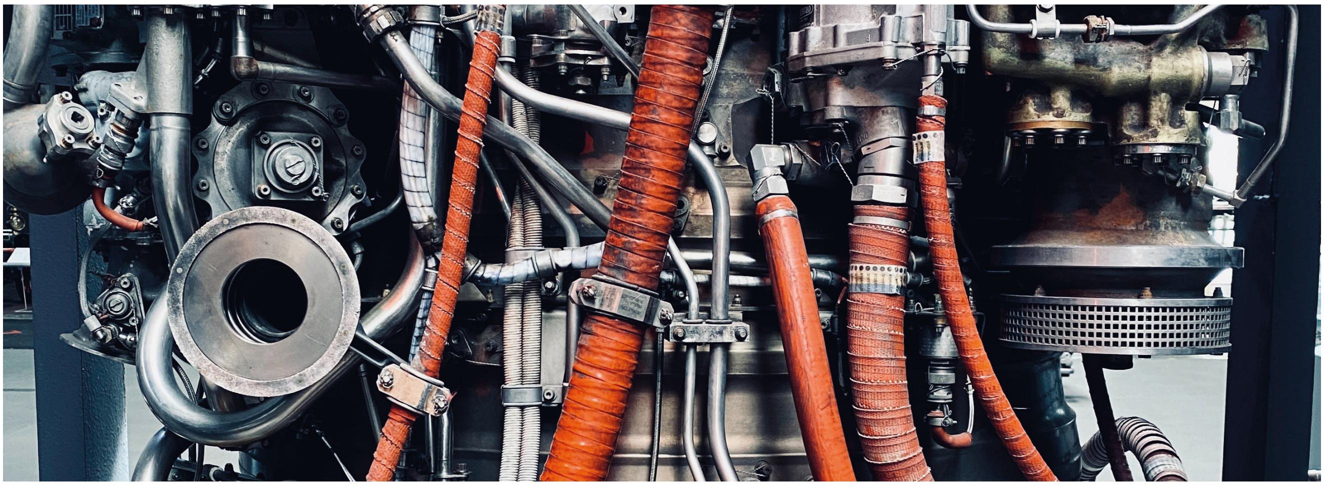


Modern embedded software is trusted to directly control critical functions of complex machines like cars and airplanes. This trust can only be established through the use of high-quality development practices and extensive testing.

Guidance such as DO-178C and ISO 26262 exists to ensure that clear, implementable steps can be taken by embedded system developers to adhere to best practices while also prescribing specific testing requirements dependent on the criticality of the system under test.

A range of commercial embedded testing tools designed to make rigorous embedded software testing more efficient and cost-effective are available. In this guide, we will detail 5 key factors to consider when choosing an embedded testing tool.

2. Reference system: FADEC



Throughout this guide, we will consider key factors in reference to an example modern embedded system; a Full Authority Digital Engine Control system (FADEC).

FADECs are one of the most critical systems in a modern aircraft. With direct permission to change aircraft behavior, they are certified to the highest DO-178C Design Assurance Level (DAL), level “A”.

Producing the necessary test evidence to certification authorities that such a system meets its requirements requires a great deal of testing effort. This is a real-world use case where advanced embedded testing tools would commonly be used.

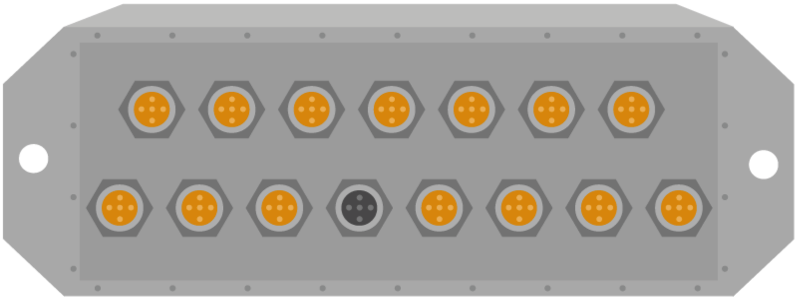
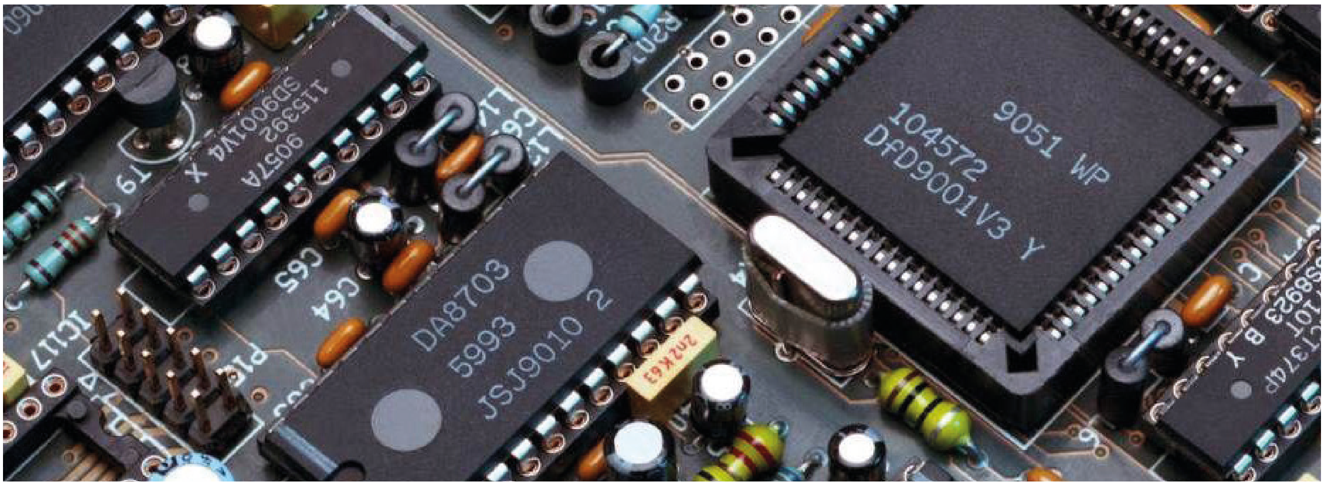


Figure 1 – FADEC model for jet engine

3. Factor #1: Flexibility



On-host software testing

In on-host testing, an application is tested on a host computer that has a different hardware environment to the final application. In on-target testing, an application is tested on the hardware to be deployed (the target).

Modern embedded developers use a wide range of development approaches, techniques and workflows. Considering how frequently these approaches evolve, it is important to choose a flexible software testing tool that adapts to how you work now and how you might work in the future.

3.1 Testing on-host and on-target

It is common practice when developing embedded systems to perform both on-host and on-target software testing.

On-host testing has several benefits, including ease of setup, the ability to start testing quickly and a lower associated cost. The main advantage for most projects is that on-host testing can be run continuously throughout the whole software development life cycle so errors can be fixed early, even when access to test rigs is not available at the start of verification.

On-host testing can also be performed off-site easily, such as by subcontracted organizations. Performing on-host testing doesn't replace the need for on-target testing, however, as software behavior often depends greatly on the hardware environment the software is hosted on. Therefore, on-target testing is needed to provide assurance that the final system behaves as expected.

When selecting an embedded testing tool, it makes sense to choose one that can perform both on-target and on-host testing. This flexibility will let you use the same tool to perform testing throughout your project. Other features to look out for include the ability to use the same test suite for both on-host and on-target testing and the ability to merge results from on-host and on-target testing. Rapita **Verification Suite** (R**VS**), for example, lets you write tests that can be run both on-host and on-target. It also allows on-host and on-target test results to be merged into a single report for easily analysis.

FADEC Example

We need to do on-target testing as the FADEC is a DO-178C DAL A development. We want to start testing as early as possible to avoid nasty surprises later in the project, but we won't be able to start on-target testing until late in the project as the hardware is still being developed. The ideal testing tool will let us create tests that will run both on-host initially then on-target later in the software verification lifecycle.

3.2 Verification activities

A number of different verification activities may be rquired to demonstrate that embedded software is reliable. These include, for example, requirements-based functional testing, structural coverage analysis and worst-case execution time analysis.

As reducing the number of tools you use will increase your efficiency, the ideal testing tool will be flexible and support as many of the verification activities you need to do as possible. For example, R**VS** supports requirements-based functional testing (Rapi**Test**), structural coverage analysis (Rapi**Cover**) and worst-case execution time analysis (Rapi**Time**) all from one integrated platform.

FADEC Example

As the project is a DO-178C DAL A project, many verification activities are needed for this level of compliance. The ideal testing tool for the project would be able to perform a range of activities, for example performing requirements-based testing, structural coverage analysis and worst-case execution time analysis. This will reduce project costs and improve efficiency as engineers can perform all activities from the same toolsuite. It will also reduce the number of qualified target integrations required, the number of qualification kits and reduce tool-learning costs.

Structural coverage

Structural coverage is a measurement of how much code is executed during testing and is a metric often used to assess the completeness of requirements-based testing.

Worst-case execution time

Worst-case execution time is the maximum length of time a task takes to execute on a specific hardware platform. WCET is a metric commonly used in reliable real-time systems which have a non-negotiable deadline for execution.

3.3 Language support

Choosing an embedded testing tool that supports a range of key embedded programming languages, and not only your “core” language is a very sensible way to de-risk your project in terms of deadlines and budgets.

While new code for a project may all be written in one “core” language, libraries, APIs or other supplemental code may be utilized in the system under test that is not written in the “core” language.

If the language support of tools is not properly considered during planning, it may be the case that supplementary codebases are not supported by the tool you choose. Introducing new tooling later in the development lifecycle to address these gaps can add significant costs and inefficiencies in your verification workflow.

Figure 2 – R**VS** supported languages

Advanced embedded software testing tools support a range of the most commonly used embedded programming languages. R**VS**, for example, supports Ada, C and C++ code. As these are the three main languages used to write embedded software, the support of these languages ensures that most projects can perform all of their on-host and on-target testing across their entire codebase using the same tool.

FADEC Example

Our system is written in C, but includes C++ libraries. For this project,a tool being able to support both C and C++ as a minimum is an important criteria when choosing a testing tool. This will ensure that the project will not require additional tools for software testing, reducing project cost and improving efficiency.

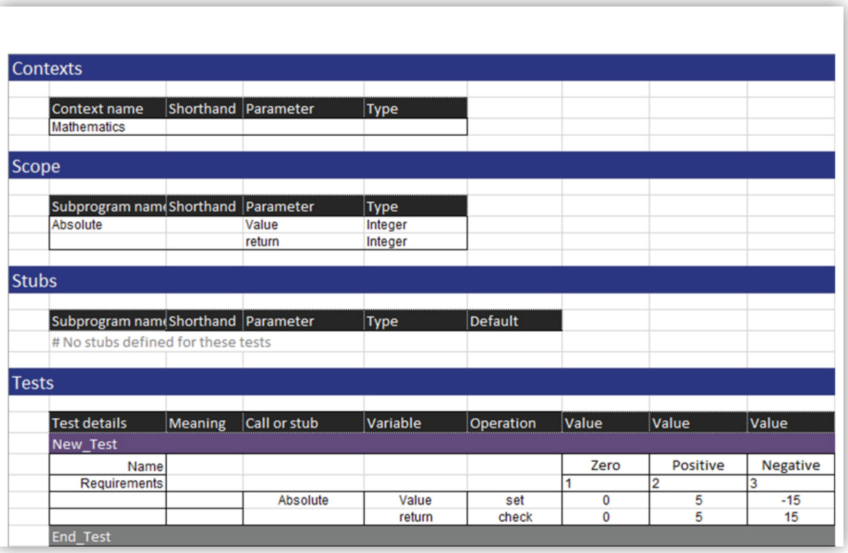
3.4 Test authoring formats

There are a variety of ways to author tests designed for embedded software. The most direct way to write tests is to use a standard programming language to define them. This can be done either by writing test code in the software itself or storing it in an external file. This is a viable testing option for small projects where engineers are programming experts, though for compliance with guidelines such as DO-178C, you would not be able to include internal test code in your final software.

Functional testing tools often include other types of testing formats that can make testing easier and more efficient. The most common formats can be categorized as GUI-based test formats, spreadsheets and platform-specific scripting languages.

- **GUI-based testing formats** – some embedded testing tools let you write tests directly in their Graphical User Interface (GUI). Where this is offered, writing tests in this way will typically be made easy thanks to helpful features such as autocompletion and automatic error checking. Another benefit of writing tests in this way is that testers should not require an in-depth knowledge of the programming language of the system under test. Drawbacks of writing tests in a GUI-based testing format include that it can be difficult to migrate such tests to another tool if this is needed in the future and that GUI-based testing formats cannot typically be easily reviewed for differences between different test revisions.
- **Spreadsheets** – spreadsheets are a popular method of test authoring as they are easy to use and highly portable. Also, spreadsheet test users should not need an in-depth knowledge of the programming language of the system under test to be able to write tests. On the downside, testers will need to understand the spreadsheet format itself. Also, some spreadsheet formats may not provide easy ways to use certain features that are easy to use with other formats, such as writing tests with conditional logic. Another downside is that spreadsheet tests cannot typically be easily reviewed for differences between different test revisions.
- **Tool-specific scripting languages** – some embedded testing tools offer custom scripting languages specifically designed to drive tests on that platform, like the RapiTest Scripting Language. As these languages are specifically designed for testing, they typically provide a feature-rich and efficient way of writing tests for engineers that can learn the language. Some downsides of using scripting languages are that engineers will need to learn the language, making it less suitable for those without programming experience, and that migration of tests written in these formats can be a concern as tests will not be directly supported by other testing platforms.

As there are benefits and drawbacks to different test formats, the best embedded software testing tools will let you write tests in a range of formats. For example, R**VS**'s functional testing plugin Rapi**Test** lets you write tests in mature and well documented spreadsheet and script formats, and a GUI-based testing format for the tool is coming in 2021.



Contexts

Context name	Shorthand	Parameter	Type
Mathematics			

Scope

Subprogram nam	Shorthand	Parameter	Type
Absolute		Value	Integer
		return	Integer

Stubs

Subprogram nam	Shorthand	Parameter	Type	Default
# No stubs defined for these tests				

Tests

Test details	Meaning	Call or stub	Variable	Operation	Value	Value	Value
New_Test							
Name					Zero	Positive	Negative
Requirements					1	2	3
		Absolute	Value	set	0	5	-15
			return	check	0	5	15
End_Test							

Figure 3 – Writing tests in R**VS** using speadsheets

FADEC Example

Some of the test engineers working on the project have little experience writing C and C++ code. Also, some of the software functions are being developed and tested by a subcontracted organization, while the main organization will be reviewing the tests and results. An ideal testing tool for this project will include test formats that don't require test engineers to know how to write C and C++, such as GUI-based test formats or spreadsheets, and tests that are easily diffable such as a tool-specific scripting language, as this will allow the main organization to easily review tests written by the subcontracted organization.

3.5 Flexible licensing

Different organizations have different team structures and workflows. The testing tool you use should fit into your current team structure rather than the other way around. Some testing tools may include licensing options that let multiple people within the same organization (or even different organizations) use the software in a shared environment by offering floating licenses. Licensing could be offered on either a subscription basis or perpetual right to use the tool, which can be a great option for long-running projects such as aerospace software development projects.

Using a tool with flexible licensing options will let you select the best option to align with your project, budget, and team structure. R**VS**, for example, lets you license software on either a node-locked (a license can only be used on specific machines) or floating (a license can be used by anyone within a team determined by the customer) basis, with either subscription-based or perpetual licensing durations.

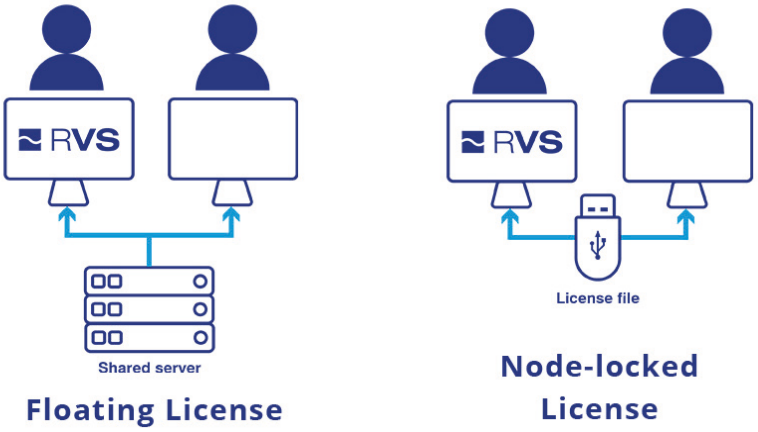


Figure 4 – R**VS** license options

FADEC Example

The main organization has multiple working sites in different time zones and wants to share licenses with subcontractors. A "floating" license can support all of these groups and will also allow the main organization to track license usage throughout the software verification process. As testers are working across multiple time zones, using a floating license can also reduce the number of licenses that need to be purchased, reducing overall testing costs. As we expect that the project will run for many years, the ideal licensing duration will be a perpetual license, reducing overall cost compared to purchasing and renewing a subscription-based license.

Licensing options

Find out more about R**VS** licensing options at:

rapitasystems.com/licensing

4. Factor #2: Interoperability

Modern software development practices

It is common for software developers to use a wide range of third-party tools to increase the efficiency of tasks such as software configuration management and continuous building and verification of code.

In contexts where requirements-based testing is critical, such as when following DO-178C, requirements management tools are often used to help manage the large number of requirements and related artifacts typical in a project.

Software development can be made easier by using third-party tools to manage things such as version control, requirements and automated testing.

These tools help to ensure that all developers are working on the latest builds, running the right tests, and that test engineers can all see the same results. Integration with such tools can save a lot of time and reduce errors in the embedded testing process.

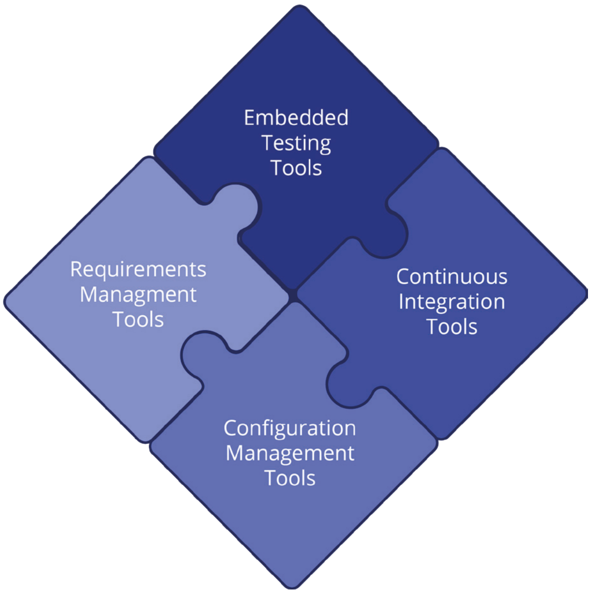

Embedded Testing Tools
Requirements Managment Tools
Continuous Integration Tools
Configuration Management Tools

Figure 5 – *Software development tools interoperability*

4.1 Testing in continuous integration environments

Continuous integration servers are often used to keep track of software development over time. When an automated test tool can interface with the continuous integration server being used, tests can be automatically run with each new revision of the software. This helps with software quality control as it lets you track your testing progress over time and identify which versions of the source code caused failures. Interoperation between your testing tool and your continuous integration tool can help you ensure that your verification results stay up to date throughout your project's entire life cycle.

If you are performing requirements-based testing, you may want to select an embedded testing tool that integrates with the continuous integration software you are using, if you are using one. R**VS**, for example, integrates with Jenkins and Bamboo, the most popular continuous integration tools on the market.

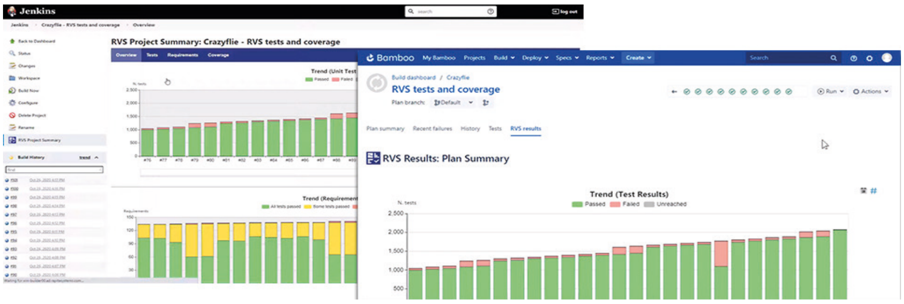


Figure 6 – R**VS** integration with Jenkins and Bamboo

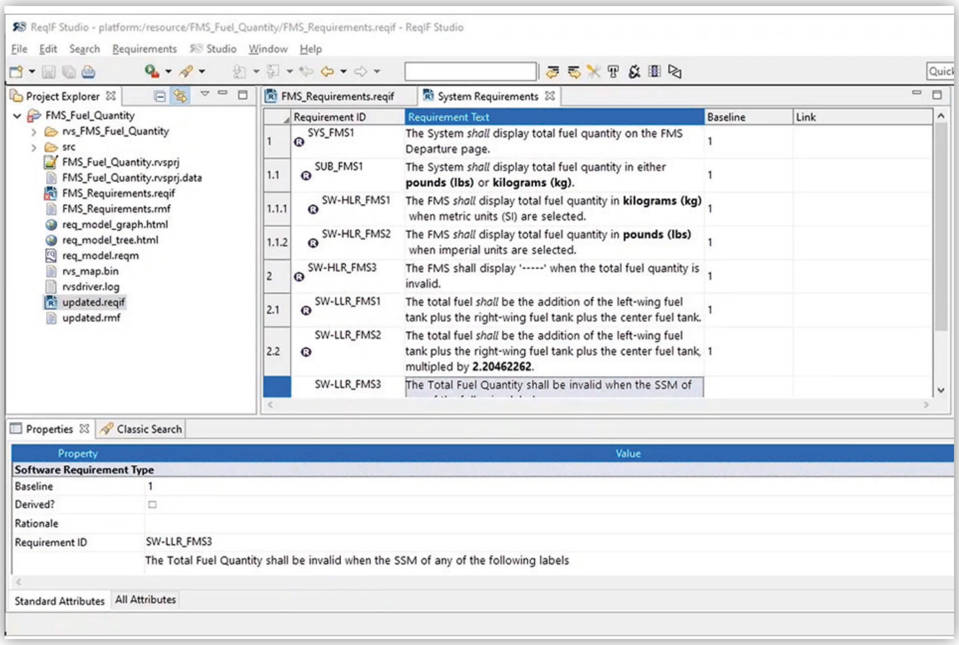
FADEC Example

Our organization is using the Jenkins continuous integration tool. Engineers at both sites are working on different sections of the code and are using a shared repository to keep up to date with the project. The ideal testing tool for the organization will be one that lets them automatically run tests with each new build and view their results within the Jenkins interface.

4.2 Requirements management

Requirements management tools help to manage the large number of artifacts that are produced and tracked in a project such as requirements, tests, and reviews in a verification project, and help maintain and track the traceability between these artifacts.

When selecting an embedded software testing tool, you may want to select one that integrates with the requirements management software you are using. R**VS**, for example, integrates with most requirements management software via the Requirements Interchange Format (ReqIF). In this way, users can manage their requirements-based testing plans by importing test results from R**VS** directly into their requirements management software.



	Requirement ID	Requirement Text	Baseline	Link
1	SYS_FMS1	The System *shall* display total fuel quantity on the FMS Departure page.	1	
1.1	SUB_FMS1	The System *shall* display total fuel quantity in either **pounds (lbs)** or **kilograms (kg)**.	1	
1.1.1	SW-HLR_FMS1	The FMS *shall* display total fuel quantity in **kilograms (kg)** when metric units (SI) are selected.	1	
1.1.2	SW-HLR_FMS2	The FMS *shall* display total fuel quantity in **pounds (lbs)** when imperial units are selected.	1	
2	SW-HLR_FMS3	The FMS shall display '-----' when the total fuel quantity is invalid.	1	
2.1	SW-LLR_FMS1	The total fuel *shall* be the addition of the left-wing fuel tank plus the right-wing fuel tank plus the center fuel tank.	1	
2.2	SW-LLR_FMS2	The total fuel *shall* be the addition of the left-wing fuel tank plus the right-wing fuel tank plus the center fuel tank, multipled by **2.20462262**.	1	
	SW-LLR_FMS3	The Total Fuel Quantity shall be invalid when the SSM of		

Property	Value
Software Requirement Type	
Baseline	1
Derived?	□
Rationale	
Requirement ID	SW-LLR_FMS3
	The Total Fuel Quantity shall be invalid when the SSM of any of the following labels

Figure 7 – R**VS** uses the ReqIF interchange format to import requirements from 3rd part tools

FADEC Example

Our organization is using Rational® DOORS® to manage requirements, test plans and reviews and support traceability between these artifacts. The ideal testing tool for our organization will be one that lets us import requirements information from Rational DOORS into our testing project and use this information when displaying and exporting results.

5. Factor #3: Efficiency

5.1 Easy startup

A well-designed and intuitive GUI makes learning and using embedded testing software easier. Good embedded testing software will include practical tutorials to help users learn how to use the software. In addition, detailed documentation will help both beginners and experts alike understand how to efficiently use the tool they've chosen. R**VS**, for example, includes a comprehensive set of practical tutorials accessible from within the GUI that make it easy to get started using the tool and writing tests, as well as comprehensive documentation.

FADEC Example

Test engineers of varying expertise will need to learn how to write and manage tests quickly. As engineers plan to use multiple test authoring formats to best meet their needs (see Test authoring formats), they will need to learn how to use these formats as well as the testing software itself. The ideal testing tool for the organization will have an intuitive user-interface and include practical tutorials and detailed documentation including language and grammar guides for any scripting formats available in the software.

Overheads

An overhead is any combination of excess computation time, memory or other resources needed to perform a task.

5.2 Low overheads

While overheads do have an effect on the efficiency of on-host testing, this effect is much more pronounced for on-target testing, where target resource limitations may require the use of multiple builds to fully test the software.

The ideal embedded testing tool will make the best use of the capabilities and limitations of each system to minimize overheads. Tools with lower overheads will let you fit more tests into each build of your code even when your target's RAM, code size or execution time is constrained, so you will need fewer builds to fully test your code. R**VS**, for example, has flexible integration libraries and an advanced Modified Condition/Decision Coverage (MC/DC) library which reduce overheads for testing software execution time and coverage.

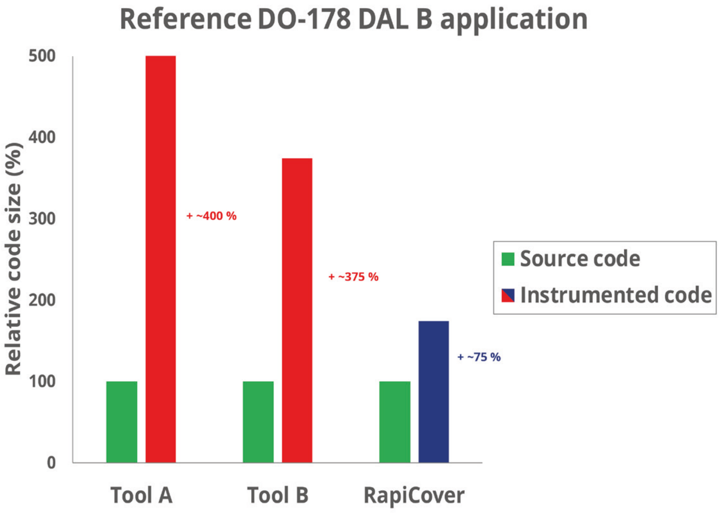


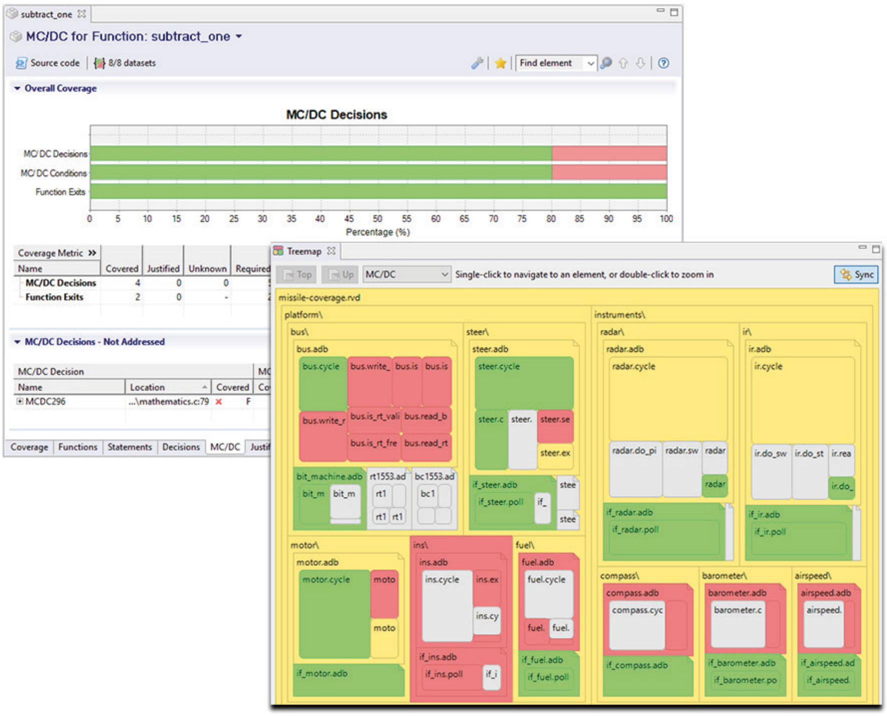
Figure 8 – Rapi**Cover**'s relative code size compared to other similair tools

FADEC Example

Our customer is requesting novel functionality to be included in the FADEC system. As is common in avionics projects, there is limited RAM and code size on the hardware on which the system will be hosted. Selecting a testing tool with low RAM and code size overheads will reduce the number of builds needed to test the software, thus increasing project efficiency.

5.3 Results analysis

Advanced GUI features can make it much easier to analyze embedded testing results. This includes the use of colored charts, filtering, sorting and searching options to locate and understand results, configurable display options to meet user preferences and accessibility needs, and configurable export formats. R**VS**, for example, includes a range of display options such as treemaps that help users understand results and filter results to show results for specific areas of the code, failed tests, or tests for specific requirements. R**VS** also enables users to export results in multiple different formats.



Coverage Metric				
Name	Covered	Justified	Unknown	Required
MC/DC Decisions	4	0	0	
Function Exits	2	0	-	

MC/DC Decision			
Name	Location	Covered	Co
MCDC296	...\mathematics.c:79 ✗	F	

Figure 9 – R**VS** Graphical User Interface

FADEC Example

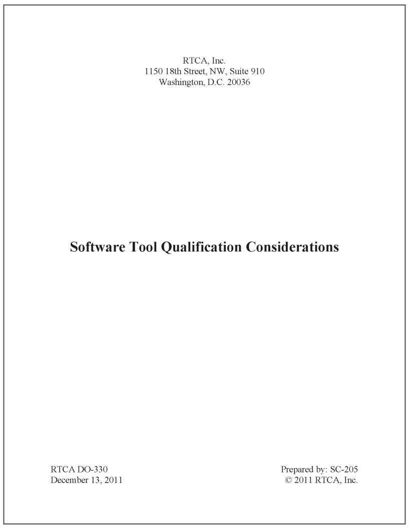
Our relatively large project will include many results and we will need to submit our test results for DO-178C certification. The ideal testing tool will include a range of options that make it easy to filter results to specific modules in the code, and will support the exporting of results into a format suitable for providing to our certification authority.

6. Factor #4: Reliability

DO-330

RTCA DO-330 is a document which provides tool-specific guidance for building airborne and ground based software. It may also be used in other domains such as automotive, space and electronic hardware.

As critical software must be robust, any testing tools used to test the software must be reliable. This is even more important when working towards compliance guidelines such as DO-178C, where evidence must be provided to demonstrate that your testing tool is robust and reliable according to "DO-330: Software Tool Qualification Considerations".



RTCA, Inc.
1150 18th Street, NW, Suite 910
Washington, D.C. 20036

Software Tool Qualification Considerations

RTCA DO-330
December 13, 2011

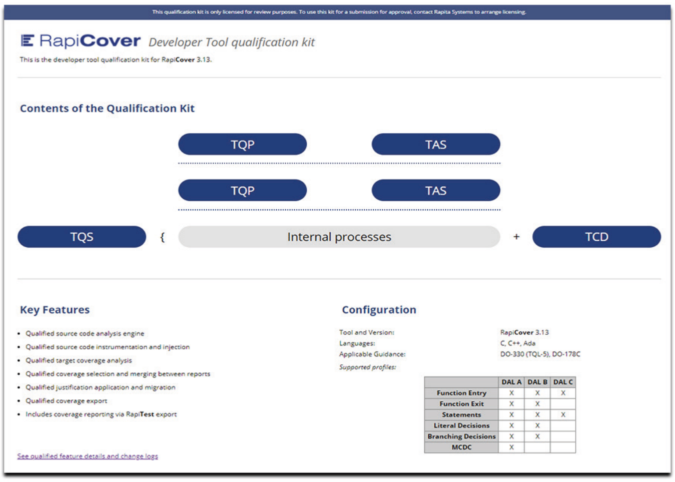
Prepared by: SC-205
© 2011 RTCA, Inc.

Figure 10 – RTCA DO-330: Software Tool Qualification Considerations

6.1 Qualification kits

The use of “Qualification kits” is the aerospace industry standard way of providing evidence that a testing tool is robust with respect to specific criteria. The gold standard set of criteria for tool qualification is listed in the “Software Tool Qualification Considerations (DO-330)” document, which can be applied to any type of software but is used most in the aerospace software domain.

The ideal testing tool will have an associated qualification kit available alongside it that can be used to demonstrate that the tool meets DO-330 considerations. For example, R**VS** tools for requirements-based testing (Rapi**Test**), structural coverage analysis (Rapi**Cover**) and worst-case execution time analysis (Rapi**Time**) have easy-to-use qualification kits available for DO-178C and ISO 26262, simplifying the certification process.



This qualification kit is only licensed for review purposes. To use this kit for a submission for approval, contact Rapita Systems to arrange licensing.

RapiCover *Developer Tool qualification kit*

This is the developer tool qualification kit for Rapi**Cover** 3.13.

Contents of the Qualification Kit

Key Features

- Qualified source code analysis engine
- Qualified source code instrumentation and injection
- Qualified target coverage analysis
- Qualified coverage selection and merging between reports
- Qualified justification application and migration
- Qualified coverage export
- Includes coverage reporting via Rapi**Test** export

See qualified feature details and change logs

Configuration

Tool and Version:	Rapi**Cover** 3.13
Languages:	C, C++, Ada
Applicable Guidance:	DO-330 (TQL-5), DO-178C

Supported profiles:

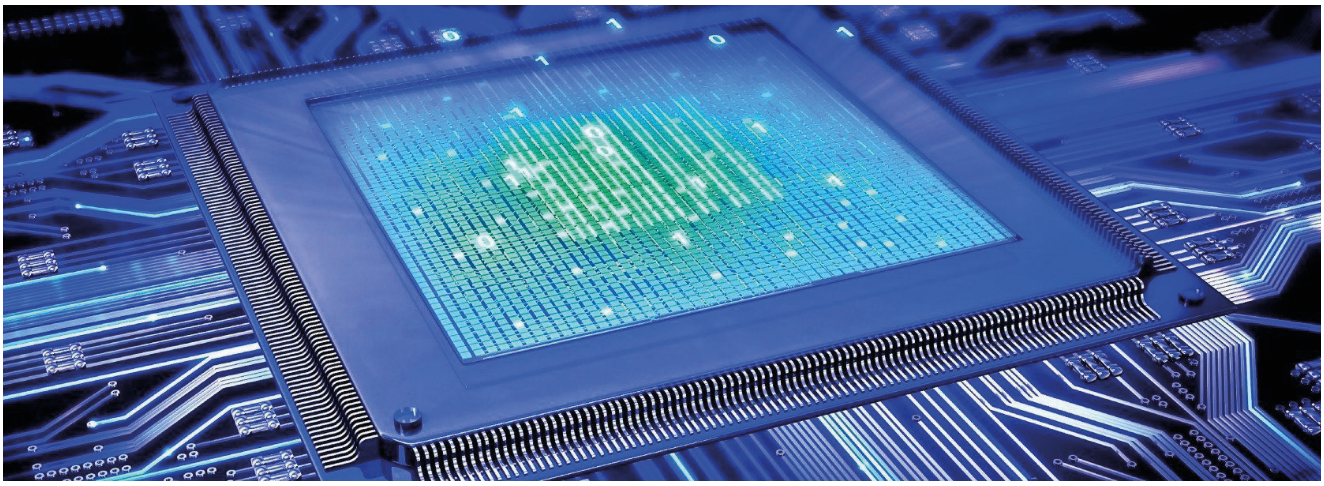
	DAL A	DAL B	DAL C
Function Entry	X	X	X
Function Exit	X	X	
Statements	X	X	X
Literal Decisions	X	X	
Branching Decisions	X	X	
MCDC	X		

Figure 11 – Sample qualification kit

FADEC Example

The software will need to comply with DO-178C at Design Assurance Level (DAL) “A”. The ideal testing tool will have an associated DO-178C qualification kit that has been used to support qualification of previous DO-178C software products to DAL A level.

7. Factor #5: Futureproof



RVS roadmap

Keep up to date with R**VS** tool development and learn about future features and enhancements.

rapitasystems.com/products/rvs/roadmap

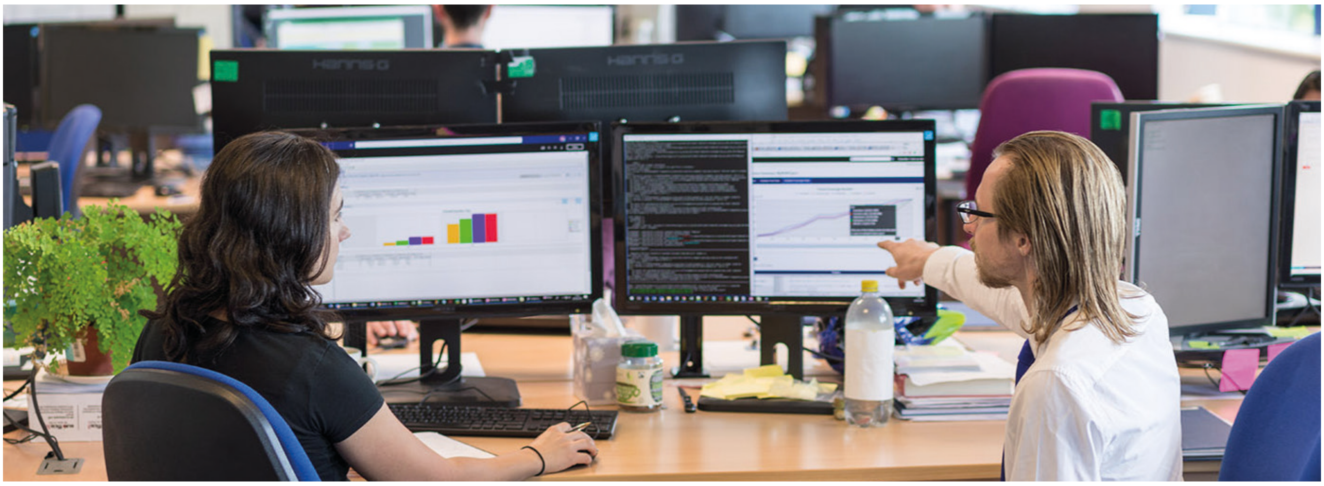
Embedded testing projects can take a lot of time, especially in the aerospace industry, so selecting an embedded testing tool is a significant commitment. When selecting your tool, ensure that the vendor will continue to support the tool for the entirety of your project plus a few more years (in case of delays). Rapita Systems, for example, offers frozen version support for R**VS**, ensuring that the version of the software you purchase will be available in the future.

You will also benefit from making sure that your tool is compatible with the most recent innovations in the embedded software industry, for example, the ability to perform on-target timing analysis testing on multicore processors. R**VS**, for example, is under constant development and has recently been improved to support the analysis of GPUs and complex multicore CPUs.

FADEC Example

Our organization has decided to use a multicore processor to meet the increasing functionality needs of the software in the project. As a DO-178C DAL A development, it will thus need to comply with the objectives identified in A(M)C 20-193, including demonstrating that the multicore software operates within its timing budgets in the presence of interference caused by contention for shared resources. The ideal testing tool will either already be able to support multicore timing analysis or have a roadmap to do so in the future.

8. Conclusion



The ideal testing tool for your embedded software testing project depends on many factors, but there are some key things to look out for:

- Flexible tools can give you more for your money and reduce the number of tools you need to use. For embedded testing tools, you should look for flexibility in terms of tools providing both on-host and on-target testing, support for different verification activities, language support, test authoring formats and licensing options.
- Interoperable tools, which work well with other third-party tools, support a smoother and more efficient testing workflow. For embedded testing tools, interoperability with continuous integration and requirements management tools are key features to look out for.
- Efficient tools can save time on your project and reduce the risk of schedule slips. For embedded testing tools, ease of startup, low on-target testing overheads and time-saving analysis features are key things to look out for.
- The reliability of an embedded testing tool is crucial, especially so for tools used in projects working towards certification such as DO-178C or ISO 26262. You should ensure that any tools you evaluate have been shown to be reliable, which is often demonstrated by the availability of qualification kits and a tool's proven history of being used to support successfully certified projects.

You can futureproof your testing environment by choosing a tool that is under continuous development to meet modern verification needs, such as the use of emerging technologies.

The Rapita **Verification Suite** (R**VS**) has been used in the critical embedded industry for over 15 years and supported a number of avionics (civil & defense) and automotive projects globally. Qualification kits for qualified R**VS** products have supported more than 20 DO-178B and C certification projects up to and including DAL A.

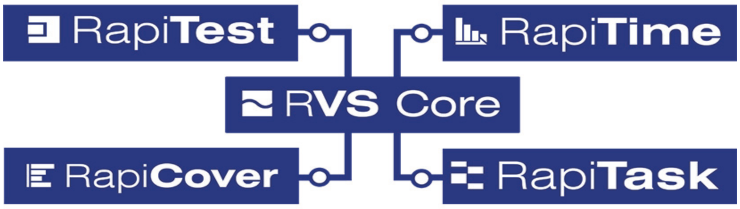


Figure 12– Rapita **Verification Suite**

Collins Aerospace

*"We use Rapita tools on some of the world's most complicated flight control systems. Compared to previous tools we've used in the past, R**VS**'s performance has been much more reliable and robust."*

Collins Aerospace® Flight Controls

Collins Aerospace® is a trade mark of Hamilton Sundstrand Corporation in United States under number 6488292. Rational® and DOORS® are trade marks of International Business Machines Corporation in United States under numbers 78086102 and 78124800.

About Rapita

Rapita Systems provides on-target software verification tools and services globally to the embedded aerospace and automotive electronics industries.

Our solutions help to increase software quality, deliver evidence to meet safety and certification objectives and reduce costs.

Find out more

A range of free high-quality materials are available at:
rapitasystems.com/downloads

SUPPORTING CUSTOMERS WITH:

Tools	Engineering Services	Multicore verification
Rapita **Verification Suite**:	V&V Services	**MACH**^{178}
Rapi**Test**	Integration Services	Multicore Timing Solution
Rapi**Cover**	Qualification	
Rapi**Time**	SW/HW Engineering	
Rapi**Task**	Compiler Verification	

Contact

Rapita Systems Ltd.
Atlas House
York, YO10 3JB
UK

+44 (0)1904 413945

Rapita Systems, Inc.
41131 Vincenti Ct.
Novi, Mi, 48375
USA

+1 248-957-9801

Rapita Systems S.L.
Parc UPC, Edificio K2M
c/ Jordi Girona, 1-3
Barcelona 08034
Spain

+34 93 351 02 05

rapitasystems.com

linkedin.com/company/rapita-systems

info@rapitasystems.com